Mi-Pal Team Description 2013

R. Coleman, V. Estivill-Castro, E. Fernandez, H. Geffner, E. Gilmore,
J. Ferrer, R. Hexel, C. Lusty, and J. Radev

ICT/IIIS, Griffith University, Nathan, QLD, 4111, Australia
DTIC, Universitat Popeu fabra, Barcelona, 08018, Spain

Abstract. The Mi-Pal team is approaching the challenges of RoboCup from a perspective of Software Engineering because the software for the robots (a) is large and complex, (b) has to be reliable, robust, and maintainable (c) involves significant diverse technologies and integration, and (d) must be developed by a team. Therefore, we have taken the approach to describe behaviour using model-driven engineering (MDE) [1]. Thus, our software development process uses models rather than programs as their principal outputs. However, while we use finite-state machines to describe actions, the declarative nature of the challenge is handled by logics. Therefore we use the paradigm of logic-labeled finite-state machines. We are able to obtain traditional architectures like, feedback-loop control, open-loop control, and behavior-oriented control. In fact, we have a hybrid agent architecture where reactive capabilities are modelled by the state-machines but the reasoning, planning and similar intelligent capabilities are obtained from inference or planning engines.

1 Introduction

Model-driven engineering is proving to be a widely successful approach for developing software. Tools and techniques are resulting in faster and simpler (easier to maintain) products and applications than traditional language parser/compiler or interpreter approaches. Model-driven engineering ensures traceability, validation against requirements, and platform independence [2]. Finite state machines in particular are ubiquitous, for instance those of executable UML [3], *MathWorks*®, *StateFlow* or *StateWorks* [4]. There are now several commercial tools and standards to represent and compose behaviours for software that will execute in embedded systems. Among others, these include `SysML` [5] and *MathWorks*^{R} *StateFlow* with *Symlink*. Penetration of these technologies includes large industrial sectors such as the automotive industry [6, 7]

The *MiPal* team has now been using logic-labeled sequential finite-state machines to describe robots behavior for several instances of the competition. However, we have a new approach as we have modernized:

1. the interpreter that now compiles such machines in a just-in time fashion, achieving exceptional efficiency on-board of the robot,
2. the inter-module communication platform, which now is a class-oriented whiteboard, and

3. the CASE tool (*MiCASE*) to design the logic-labeled finite-state machines (LLFSMs).
4. We have now developed a method to significantly reduce the verification complexity of inter-dependent state machines, making formal verification feasible for much larger sets of state machines and thus, more complex behaviour[8].
5. We also have now integrated several planners into our architecture.
6. Finally, we have modernized several aspects of our vision module.

These tools and facilities could be considered alternatives to Aldebaran tools and infrastructure, In particular, our *MiCASE* in combination with the LLFSMs could be a replacement of Coreographe. The advantage here is that our LLFSMs and their code is compiled `C++` that runs much closely to the DCM cycle (as opposed as previous LLFSMs which were interpreted). We can produce models that can be formally verified by standard model-checking tools. This is a result of the very clear semantics of ringlet execution and the very clear relationship with machines and submachines. We also have a very clear scope for all shared variables, external variables (whiteboard variables) and internal variables. The *MiCASE* tool enables very flexible designs, and provides very clear feedback to the designer while enabling any `C++` construct to be placed as code in the section of states. The whiteboard communication module is orders of magnitudes faster than the inter-process communication infrastructure provided by Aldebaran's use of `soap`.

2 Model-Driven Engineering for 2013

We use the semantics and sequential scheduling of logic-based finite-state machines (LLFSMs) [9, 10]. These LLFSMs consist of a set S of states and a transition table $T : S \times E \to S$. There is an initial state $s_0 \in S$, and for each state, the transitions leading out of the state are ordered in a sequence. Transitions are labeled by an expression $e \in E$, and these expressions are evaluated in deterministic order (and time) by an expert system. The examples in the literature use Decisive Plausible Logic (DPL) [9, 10], but the expressions can also be Boolean expressions of an imperative programming language such as `C`, `C++`, or `Java` (or any decidable logic, that provides an answer in predictable time). For RoboCup-2013, transitions are now any `C++` expresion, this is because, as we will explain later, our tools to design, build and execute LLFSMs are compiled with `clang`.

The point is that execution of a vector of these machines is sequenced deterministically by a pre-defined schedule. Each machine in the vector receives a pre-defined number of *ringlets* it executes before passing the thread control to the next machine in the vector. The execution token passes back to the first machine after the last machine completes its allocated ringlets. A ringlet consist of evaluating the **OnEntry** section of the current state (if it is the first time control arrives to this state from another state in this machine), followed by evaluation of the expressions in the list of transitions until an expression evaluates to true. When an expression evaluates to `true`, its transition fires, the **OnExit** section is evaluated and the ringlet concludes. If the list of transitions is exhausted without

any expression becoming true; then the **Internal** section of the state completes and the ringlets also conclude. Thus a ringlet is the complete assessment of the current state.

The shared variables between the different modules (LLFSMs) are called external variables and are managed on a repository architecture named the whiteboard [11]. When the execution token arrives at a machine, it makes a local copy of any external variables it will use in the current state. We refer to this as the *READ* footprint on the whiteboard. Before the execution token of an LLFSM is handed back, the machine copies to the whiteboard any external variables it has modified locally. We refer to this as the *WRITE* footprint of the state. This ensures there is never a race condition between the LLFSMs that are running concurrently under the predefined schedule (and thus, there is no need for further mechanisms to protect shared variables or synchronise LLFSMs).

For a LLFSM, the union of all the *READ* footprints of its states is called the *REQUIRES* set of the LLFSM. Similarly, the union of all the *WRITE* footprints of its states is called the *PROVIDES* set. Note that it has been shown that the *REQUIRES* set and the *PROVIDES* set of an LLFSM can be computed from the static analysis of the LLFSM description [12].

An example of a finite-state machine that implements a reactive controller for tracking the ball appears in Fig. 1. This machine has an initial state called

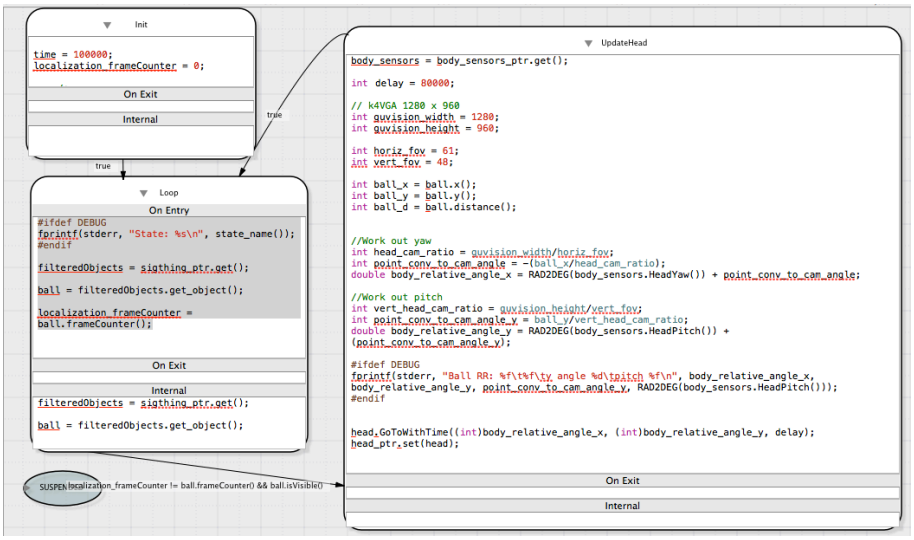

Fig. 1: A two-state LLFSM that implements a reactive control to track the ball.

`Init`. And a transition with the Boolean expresion `true` out of this state to the state named `Loop`. The state `Loop` collects information from our Kalman filter. This informaation describes where (in the frame of vision) is the ball by using

our class-oriented whiteboard to obtain a ball message (which is an object). The `UpdateHead` state performs the analysis of where to issue a motion command which is also placed in the whiteboard.

We would like to emphasize that we can place any standard `C++` code in the corresponding sections of the state of a logic-labeled finite state machine. For example, Fig. 2 shows that we can use compiler directives, and even `C` functions like `fprintf`.

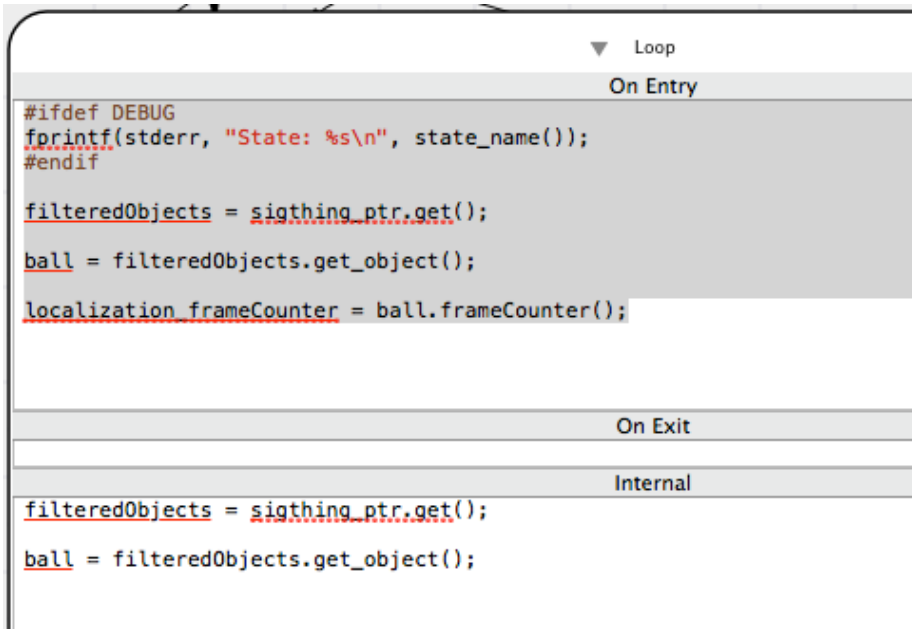

Fig. 2: The section of the state `Loop` of the logic-labeled finite-state machine of Fig. 1.

3 The class-oriented whiteboard

The whiteboard is a paradigm of module communication that enables decoupling as each module just needs to know how to post messages on the whiteboard or how to get messages from the whiteboard. We also enable modules to *subscribe* to the whiteboard and be informed each time a poster releases a new message of the subscribed message-type into the whiteboard.

The previous whiteboard was extremely flexible because it was completely untyped. In a sense, any two modules could create/design the message types that the sender would post and that the receiver would subscribe to messgages. The receiver can also inpect the whiteboard and retrieve using a `get`. Such flexibility

came with the price that there was essentially no type checking at compilation time and also the sender and the receiver had very complicated code to build or to parse messages that hold several properties or attributes of information.

Therefore, the whiteboard API has been redefined and re-implemented to allow the definition of classes that stipulate message types.

3.1 Simple posting to the whiteboard

In order to create messages you must define the class of these messages. We require, but we do not enforce, that the class implements three methods.

1. `void from_string(const std::string &str)` This method should parse and materialize an object of your class from a string. Typically, these are the list of attribute values separated by comas.
2. A constructor that takes a `std::string` as a parameter and typically uses the `from_string()` method above, along with an (optional) initialiser list to initialise its content.
3. `std::string description()` This method should serialize the object in a way that the earlier method could materialize it; while maintaining a readable format for debugging.

These three methods are important to impersonate a module that post these messages. Therefore, one is able to construct test cases of a module even if the module that produces the input is not available.

An example of constructing a class is provided in Fig. 3.

For some well known message types, their classes have already been predefined. For the following examples we will use the class of Fig. 3 and also a predefined message type `Print`.

Thus, to post a message with `Print` message-type you will need some preliminaries. These are the corresponding includes.

```
#include "gugenericwhiteboardobject.h"
#include "guwhiteboardtypelist_generated.h"
```

It is also useful to use the corresponding namespace

```
using namespace guWhiteboard;
```

Then, to post a message into the new class-oriented whiteboard, we just need to create an object (using a known whiteboard type).

```
Print_t print;
```

That is we append `_t` in order to have access to an instance on the whiteboard.

Now, we can use a setter to actually post a message with content a string.

```
print.set("We are about to loop for ever");
```

Using the class from Fig. 3 the following code posts an object of the given class.

```
Point2D point(5,3); // here is the object
BallBelief_t ball; // the pointer to access the class-based whiteboard
ball.set(point); // you use it, to post your object by using set
```

```
#ifndef Point2D_DEFINED
#define Point2D_DEFINED

#include <cstdlib>
#include <sstream>
#include <gu_util.h>

namespace guWhiteboard
{
        /** * Class for for demonstrating OO-messages.  */
        class Point2D
        {
            PROPERTY(int16_t, x) //  x-coordinate
            PROPERTY(int16_t, y) //  y-coordinate

        public:
            /** designated constructor */
            Point2D(int16_t x = 0, int16_t y = 0):
                _x(x), _y(y)  { /* better than set_x(x); set_y(y) */ }

            /** string constructor */
            Point2D(const std::string &names) { from_string(names); }

            /** copy constructor */
            Point2D(const Point2D &other): _x(other._x), _y(other._y) {}

            /** convert to a string */
            std::string description()
            { std::ostringstream ss;
              ss << x() << "," << y();
              return ss.str();
            }

            /** convert from a string */
            void from_string(const std::string &str)
            {
                std::istringstream iss(str);
                std::string token;
                if (getline(iss, token, ','))
                {
                    set_x(  int16_t(atoi(token.c_str())));
                    set_y(0);
                    if (getline(iss, token, ','))
                    { set_y(int16_t(atoi(token.c_str())));
                    }
                }
            }
        };
}

#endif // Point2D_DEFINED
}
```

Fig. 3: A simple class to post 2D-points into the whiteboard.

3.2 Subscribing to the whiteboard

The earlier section shows what a module that acts as a poster does. If you are constructing a module that is to receive each of these postings, you may choose to subscribe to the messages. For this, you need to create the call-back function that will be started in a thread every time there is a posting. Your module will need an instance variable of the type pointer to `whiteboard_watcher` .

```
whiteboard_watcher *watcher;
```

We show here two call-backs (that have the same class and thus data type). One to illustrate that we can use the same call-back for more than one message type. The other one shows dedicated call-backs for a message type.

The subscription is best if it happens in the code of the constructor of the receiver module. So the module `HelloWorld` declares in its `.h` file the instance variable for the whiteboard `watcher`.

```
watcher = new whiteboard_watcher();
```

Once we have this, we can use it in the constructor of `HelloWorld.cc` to subscribe.

```
SUBSCRIBE(watcher, Print, HelloWorld, HelloWorld::callback);
SUBSCRIBE(watcher, Say, HelloWorld, HelloWorld::callback);
SUBSCRIBE(watcher, QSay, HelloWorld, HelloWorld::callback2);
```

Here, we are using the same call-back for two message types: `Print` and `Say`. In the call back itself you can retrieve the message type in the first parameter, while the content is the second parameter. You can note that in Fig. 4 the second parameter is properly typed.

```
void HelloWorld::callback(WBTypes t, string &stringToPrint)
{ // NOTE: we are distinguishing the message types here
  if (t == kPrint_v)
    cout << "Print Message type " << t << " is '" << stringToPrint << "'";
  else if (t == kSay_v)
    cout << "Say Message type " << t << " is '" << stringToPrint << "'" << endl;
  else
    cout << "Unknown Message type " << t << " is '" << stringToPrint << "'" << endl;

  cout << endl;
}

void HelloWorld::callback2(WBTypes t, string &stringToPrint)
{  cout << "QSay Message type " << t << " is '" << stringToPrint << "'" << endl;
}
```

Fig. 4: The code of the call backs in the class `HelloWold.cpp`

4 *MiCASE*

As previously mentioned, *MiCASE* is a tool developed to visually design logic-labeled finite state machines. We previously used a freely available tool called `qfsm` [1]., and created our own version which enabled tracking of the execution of FSM's remotely, using our existing Whiteboard infrastructure. The expansion of `qfsm` to a tool that enabled tracking of the execution of a LLFSM was illustrated in the classification video (and also in a simulation demonstration `http://youtu.be/y4muLP0jA8U`). We have also demonstrated the the value of simulation [13]. Despite the availability of the source code for `qfsm`, rather than continue to modify `qfsm`, we have decided to create a new tool called *MiCASE* which has been developed with our specific LLFSMs usage in mind.

In the past, The language used to describe the actions in each state used to be a 'C-like' language we manually specified and parsed using ANTLR (`http://www.antlr.org`). We found some drawbacks of using this parsing system; the most critical being that we found it difficult to manage how ANTLR allocated memory and suffered from performance loss on the Naos. The second drawback was that new language features had to be added manually to the grammar, and it hindered development when a feature was missing. The *MiCASE* editor and the `CLFSM` interpreter have done away with the 'C-like' language and now use standard `C++` to describe actions within states. Instead of interpreted actions, we have binary executables compiled with `clang`. *MiCASE* converts states containing fragments of `C++` code into header and implementation files that work with `CLFSM` and run on the Nao.

MiCASE has been developed from the ground up to aid design of `C++` LLFSMs'. A distinguishing feature is GUI support for including `C++` header files and defining variables, at the machine and state level. Fig. 5 shows the versatility of *MiCASE*for designing LLFSMs and to display the different variables (externa, lcoal and itnernal).

5 Planning

Our whiteboard architecture allows very flexible constructs; and in particular, it is easy to create behaviors based on feedback-loop control, open-loop control, reactive systems, hybrid-reasonings architectures, and behavior-based control. We have been able to incorporate the large body of knowledge from planning. In fact, *planners* have become standard pieces of software that we smoothly integrate into our LLFSMs.

Because now inputs to a domain-independent planer can be provided in the Planning Domain Definition Language PDDL [14–16], we integrate different planers as long as they conform to such standardized planning language. We also need to integrate the standardized output so that the resulting fully-ordered or partially-ordered sequence of actions is transmitted to the other elements of the architecture that execute them.

[1] qfsm.sourceforge.net

Fig. 5: A rotated view in *MiCASE* of the two-state LLFSM of Fig. 1.

5.1 The "plan, execute, re-plan" cycle

The use of a planner in a robot can be organized in a cycle where the robot control provides a planner with a *planning request* (usually consisting of the robot's current state, and a goal state). Sensing is involved in checking the success of each action while executing the plan, if the action fails (as perceive by the sensors), the belief the robot has for its current state is used in a new planning request (re-planning stage) and the robot moves into execution again. Under the name of *reactive executive monitoring* [17] a reactive planning engine [18] monitors the systems low-level states against a declarative model of the robot's functionality, while continuously performing sense-plan-act cycles [19–22]. In the planing stage, the reactive planner finds a plan (a sequence of robot actions) under some parameters; which may include a planning horizon. Such a parameter balances long planning times with a reactive behavior. If a robot is operated with monitoring, the belief of the robot can be contrasted with operator directives and enable supervised autonomy [23] and integration of intervention by a human operator [24].

We now illustrate how our architecture constructs this paradigm. We provide a module that enables several planners to be integrated with the whiteboard architecture through the following API.

LOAD_PLANNER(***a planner***) : This enables selecting a planner. In our prototype implementation we currently can chose between a *regression planner* or the `Lama` planer [25]. The planning module posts to the whiteboard either success (a known planner was provided) or failure.

LOAD_PDDL(***name of problem description***) : This enables the planner to retrieve and load a planning problem in PDDL. If there is no file with that name or it does not conform to PDDL, an error is posted to the whiteboard; otherwise, success is reported.

START_PLANNER(***depth***) : This starts the planner with a certain maximum depth of actions (a horizon). The *depth* parameter is optional, and if not supplied, plans of any finite depth are sought. The planner constructs a plan. If a plan exists, confirmation is posted to the whiteboard. Failure is reported when there is no sequence of actions from the source to the goal.

NEXT_ACTION(***rank***) : This request the *rank*-th action in the current plan. Note that the planer responds with also the action numbered (and the action parameters). This provides some robustness to lost exchanges when we distribute the whiteboard over a network using UDP.

RE-PLAN(***source, depth***) : This planer finds a new plan like START_PLANNER, but from a new source.

IS_OBSTACLE_KNOWN(***position***) : This reports to the planner an obstacle at the supplied *position*. This may be an obstacle found along the way or an explanation for why the last action failed. The planner responds weather this obstacle was detailed in the problem description or if it is an obstacle the planer was not aware of. In the later case, the planner updates the problem description to now include the obstacle.

Our API provides other tools to convert formats and description of plans or planning problems, but for the purposes of this report this shall suffice. Also this API allows planing in an environment where the original map (or the problem description) does not have all obstacles (and the robot may discover new ones during execution). New obstacles may trigger re-planing. Fig. 6 illustrates the LLFSMs exactly as used in our robots, and in the spirit of Model-Driven Engineering that is used to execute the "plan, execute, re-plan" cycle for our examples with two robotic platforms navigating an environment with some known obstacles but also some unknown obstacles. We omit the few initialization states and transitions for selecting a planner and loading the problem description for simplicity of the figure. We note that the LLFSM also interacts with the motion module by issuing the motion commands and then receiving from the motion module whether the motion completed or an obstacle was found. We empha-

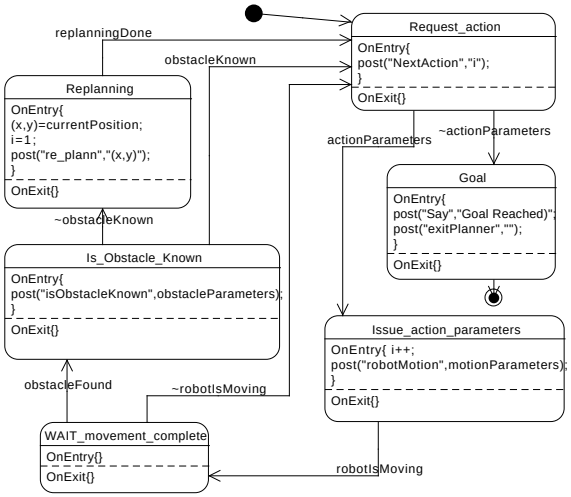

Fig. 6: The LLFSMs interpreted (and also compiled) used to run the plan, execute, re-plan cycle in our experiments.

size that the user develops the behavior graphically and directly as presented in Fig. 6. Besides the classification video, a video of this integration that also illustrates the platform independence of our LLFSMs and the planers is a planning demonstration using Webots (`http://youtu.be/-mvppFPWfMU`).

6 Vision

6.1 *Vladcal*

Vladcal is the tool used to calibrate vision on the NAOs. It creates PART color classifiers[26, 27]. This tool is used for manually selecting pixels from existing images; either by selecting rectangular regions or by selecting regions delimited by a Sobel margins. *Vladcal* has been modernized in order to use the last version of the GUI libraries (SWING) and to incorporate new functionalities for new developments. This tool is built in `java` in order to be portable to different platforms. One of the improvements is a window where the user can undersample a class when the user is working with a highly unbalanced training set. By sliding the horizontal bars the user can choose to use only a part of the instances of a certain class. The window than enables this is show in Fig. 7. Another new

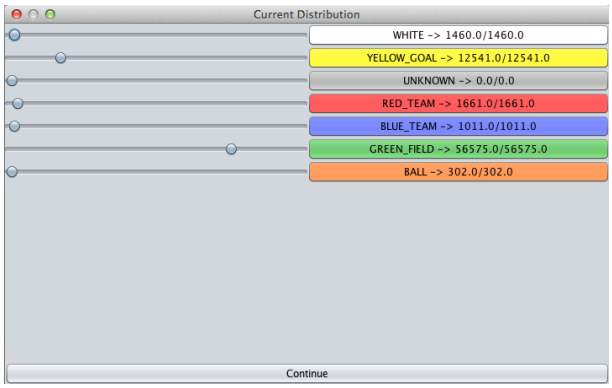

Fig. 7: Sliding bars that allow to distribute proportions of classes in training examples.

functionality is to create specialized classifiers like one-vs-all, or two-vs-all. On that way we implemented this is to build a classifier that only knows the green of the field and the white of the labels against all the other colours. t is easy to build these classifiers as users only have to select the colors that the wish to classify from a multi-select list against the rest (the unselected). The menu than enables this is shown in Fig. 8 With the Modern *Vladcal* it is possible to open and work with jpeg images, which is a widely more popular format than the previous ai2 used by NAOs and inherited from the time of the Sony AIBO. By including jpeg, Modern *Vladcal* allows working with images that other tools also use, such as MATLAB. It is also possible to connect to a NAO by introducing its IP or its Zero-conf address, and to retrieve an image on demand in real time through streaming in order to get instances of the different colours of the actual environment.

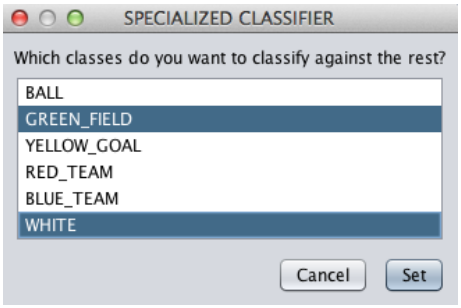

Fig. 8: The menu to select classes for the decision list classifier, so it can be tailored even using the same training file.

Some new capabilities have been added to *Vladcal* for the development of shape-based classifiers. For that purpose the Modern *Vladcal* allows to save subimages with the shapes that the user wants to classify by selecting them manually and to generate randomly picked subimages that are negative instances. It also allows computing the average size of the positive subimages taken. Fig. 9 shows selection of regions to find Naos as positive shapes.

6.2 Line recognition with RANSAC

In the past we were using OpenCV and the Hugh transform to identify lines in an image in order to find the lines in the field. Our implementation proved to be more efficient. Line recognition now runs as part of the the dlc loop in the soccer pipeline, so all the is required to start it is is to run vision. When lines are identified they will be posted to the whiteboard. The message content is of the format startx, starty, endx, endy, length, startx... Because of the complexity of the RANSAC algorithm it can on some occasions reduce the frame rate bellow the desired 30fps. There are two #define statements in guvision_dlc.cc that greatly affect both the speed and accuracy of the lines detected by RANSAC. The first parameter is MAX_TRIALS this determine how many times the RANSAC algorithm will run for each line and is currently set at 30, which appears to be a reasonable balance between speed and accuracy. Setting this parameter higher then this will result in lines that are "centered" slightly better; however the performance cost is high. The second parameter is called RANSAC_SKIP, this affects how many pixels are used in the RANSAC algorithm. Although the image is already scaled down, it was found that relatively few pixels were need for the RANSAC algorithm to work effectively, because of this the skip parameter which is currently set at 10 was introduced. This simply means that for each pixel classified in the scaled down image only every 10th one is used for line detection, it may be possible to increase this parameter further and still achieve accurate results.

6.3 Shape recognition

One of the main aspects of the RoboCup that shall evolve each year is the need for color coding. For instance, in 2012 goals of the same color were adopted. However, other color restrictions have not been modified. The field remains green with white lines on it, the colors of the teams are predefined cyan and magenta and the ball continues to have no pattern and only orange. It would be inter-

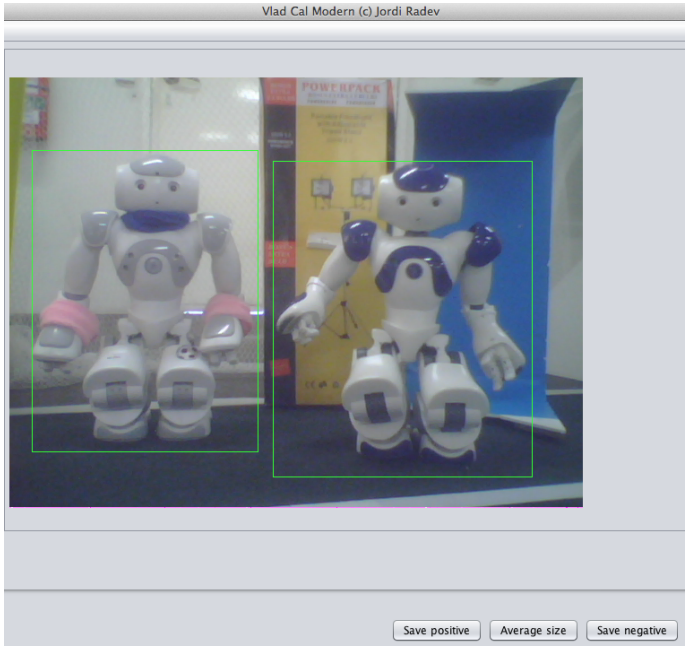

Fig. 9: The possibility to select objects as positive training examples for a recognition of shapes.

esting to remove these restrictions. Once we overcome these restrictions, each team would be able to wear its own uniform with only small color limitations to distinguish from the opponents uniform. That would mean that at the beginning of the game each robot would have to learn which are its opponents colors. As a consequence, we could have matches where the color of the ball, or even the goals color is not pre-determined. Assuming that colors of objects are no longer predetermined means that our previous knowledge of colors is not reliable for the match. A new parameter is needed for objects recognition; and we propose to use shapes. However, computer-vision techniques for shape recognition are, computationally, much more CPU-intensive than color recognition and perhaps they are unaffordable during the game. For this reason, we are currently developing a process that consists on identifying RoboCup objects on the basis

of their shape and learning autonomously the colors of the recognized objects. The chosen shape recognition algorithm is the Histogram of Oriented Gradients; this method has been proved to be capable to recognize complex objects. Actually, this algorithm is widely used for pedestrian recognition. The most complex objects we want to recognize are other NAOs, similar in complexity for their anthropomorphic shape but less variable than human beings.

At first the process will recognize the field, the lines and the ball colors in order to learn the environment set of colors. After that the process will recognize the NAOs and their team shirts to extract the colors of the shirts and to learn the actual team colors of the actual game. Finally a classifier will be built automatically to be able to recognize the objects by their colors in the environment. When the NAO has built its color classifier it will play a soccer game in order to prove that it has learnt the context and is able to play normally. Fig. 10 shows our implementation of HOG to recognize standing up Naos.

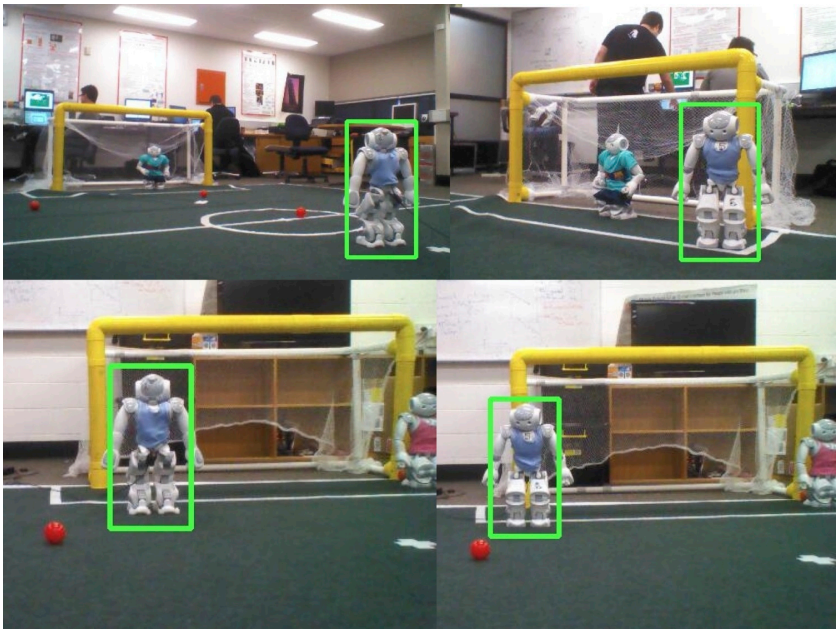

Fig. 10: Standing Nao's recognized using HOG.

References

1. Sommerville, I.: Software engineering (9th ed.). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2010)
2. Schmidt, D.: Model-driven engineering. IEEE Computer **39**(2) (2006)

3. Mellor, S.J., Balcer, M.: Executable UML: A foundation for model-driven architecture. Addison-Wesley Publishing Co., Reading, MA (2002)
4. Wagner, F., Schmuki, R., Wagner, T., Wolstenholme, P.: Modeling Software with Finite State Machines: A Practical Approach. CRC Press, NY (2006)
5. Friedenthal, S., Moore, A., Steiner, R.: A Practical Guide to SysML: The systems Modeling Language. Morgan Kaufmann Publishers, San Mateo, CA (2009)
6. SLSF, M.A.: Modelling design and style guidelines for the application of Simulink and Stateflow. The Motor Industry Software Reliability Association, Warwickshire, UK (2009)
7. GMG, M.A.: Generic modelling design and style guidelines. The Motor Industry Software Reliability Association, Warwickshire, UK (2009)
8. Estivill-Castro, V., Hexel, R.: Module isolation for efficient model checking and its application to FMEA in model-driven engineering. In: ENASE 8th International Conference on Evaluation of Novel Approaches to Software Engineering. (July 2013) to appear.
9. Estivill-Castro, V., Hexel, R., Rosenblueth, D.A.: Efficient modelling of embedded software systems and their formal verification. In Leung, K.R., Muenchaisri, P., eds.: The 19th Asia-Pacific Software Engineering Conference (APSEC 2012), Hong Kong, IEEE Computer Soceity, Conference Publishing Services (4th - 5th December 2012) 428–433
10. Estivill-Castro, V., Hexel, R., Rosenblueth, D.A.: Efficient model checkign and FMEA analysis with deterministic scheduling of transition-labeled finite-state machines. In Wang, P., ed.: 2012 3rd World Congress on Software Engineering (WCSE 2012), Wuhan, China (6th-8th November 2012) 65–72
11. Hayes-Roth, B.: A blackboard architecture for control. In Bond, A.H., Gasser, L., eds.: Distributed Artificial Intelligence, San Francisco, CA, USA, Morgan Kaufmann Publishers Inc. (1988) 505–540
12. Estivill-Castro, V., Hexel, R.: Module interactions for model-driven engineering of complex behavior of autonomous robots. In Lavanzza, L., Fernandez-Sanz, L., Panchenko, O., Kanstren, T., eds.: The Sixth International Conference on Software Engineering Advances. ICSEA 2011, Barcelona, Spain, IARIA (October 23-29 2011) 84–91
13. Coleman, R., Estivill-Castro, V., Hexel, R., Lusty, C.: Visual-trace simualtion of concurrent finite-state machines for valdiation and model-checking of complex behavior. In Ando, N., Brugali, D., Kuffner, J., Noda, I., eds.: SIMPAR 3rd Int. Conf. on Simulation, Modeling and Programming for Autonomous Robots. Volume 7628., Tsukuba, Japan, Springer-Verlag Lecture Notes in Computer Science (November 5th-8th 2012) 52–64
14. McDermott, D.: The 1998 AI planning systems competition. AI Magazine **21**(2) (summer 2000) 35–56
15. Fox, M., Long, D.: PDDL2.1 : An extension to PDDL for expressing temporal planning domains. Journal of Artificial Intelligence Research **20** (2003) 61–12
16. Fox, M., Long, D.: Modelling mixed discrete-continuous domains for planning. Journal of Artificial Intelligence Research **27** (2006) 235–297
17. Carbone, A., Finzi, A., Orlandi, A., Pirri, F.: Model-based control architecture for attentive robots in rescue scenarios. Autonomous Robots **24** (2008) 87–120
18. Beetz, M., McDermott, D.V.: Improving robot plans during their execution. In: Proceedings of artificial intelligence planning systems, Menlo Park, AAAI Press (1994) 7–12

19. Musliner, D.J., Durfee, E.H., Shin, K.G.: CIRCA: A cooperative intelligent real time control architecture. IEEE Transactions on Systems, Man, and Cybernetics **23**(6) (1993) 1561–1574
20. Williams, B.C., Nayak, P.P.: A reactive planner for a model-based executive. In Pollack, M., ed.: Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI-97), San Mateo, CA, Morgan Kaufmann Publishers (1997) 1178–1185
21. Muscettola, N., Dorais, G.A., Fry, C., Levinson, R., Plaunt, C.: IDEA: planning at the core of autonomous reactive agents. In: Proceedings of NASA workshop on planning and scheduling for space. (2002)
22. Finzi, A., Ingrand, F., Muscettola, N.: Model-based executive control through reactive planning for autonomous rovers. In: IROS. (2004) 879–884
23. Haigh, K.Z., Veloso, M.M.: Interleaving planning and robot execution for asynchronous user requests. Autonomous agents (1998) 79–95
24. Finzi, A., Orlandini, A.: Human-robot interaction through mixed-initiative planning for rescue and search rovers. In: AIIA. (2005) 483–494
25. Richter, S., Westphal, M.: The LAMA planner: Guiding cost-based anytime planning with landmarks. Journal of Artificial Intelligence Research (JAIR) **39** (2010) 127–177
26. Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., Witten, I.H.: The WEKA data mining software: an update. SIGKDD Explorations **11**(1) (2009) 10–18
27. Bouckaert, R.R., Frank, E., Hall, M., Kirkby, R., Reutemann, P., Seewald, A., Scuse, D.: WEKA Manual for Version 3-6-2. The University of Waikato (2010)